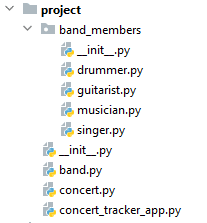
# Python OOP Exam - Concert Tracker App

**Submit your solutions in the SoftUni judge system at** [**https://judge.softuni.org/Contests/Practice/Index/3622#0**](https://judge.softuni.org/Contests/Practice/Index/3622#0)

*Ah… music. One of the best ways to enjoy life. Have any favorite bands? Today we will be in charge of creating a concert tracker app that will create bands, members, and concerts.*

You will be provided with a **skeleton** that includes all the folders and files that you will need.

***Note: You are not allowed to change the folder and file structure and change their names!***



**Judge Upload**

For the **first two problems**, create a **zip** file with the **project** **folder** and **upload it** to the judge system.

For the **last problem**, create a **zip** file with the **test folder** and **upload it** to the judge system.

You do **NOT** need to include **in the zip file** your **venv**, **.idea**, **pycache**, and **\_\_MACOSX** (for Mac users), so you do not exceed **the maximum allowed size** of **16.00 KB**.

# Structure (Problem 1) and Functionality (Problem 2)

Our task is to implement the **structure and functionality** of all the classes (properties, methods, inheritance, abstraction, etc.)

You are **free to add additional attributes** (instance attributes, class attributes, methods, dunder methods, etc.) to simplify your code and increase readability as long as it does not change the project's final result according to the requirements and the program works properly.

## 1. Class Musician

In the **musician.py** file, the class **Musician** should be implemented. It is a **base class** for any **type of musician,** and it **should not be able to be instantiated**.

### Structure

The class should have the following attributes:

* **name: str**
* A string that represents the **name of the musician**.
* If the name is **an empty string or contains only white spaces**, raise a ValueError with the message: **"Musician name cannot be empty!"**
* **age: int**
* An integer that represents the age of the musician.
* The musician must be **at least** **16 years old**; if not - raise a **ValueError** with the message **"Musicians should be at least 16 years old!"**
* **skills: list**
* An **empty list** that will contain all **skills a musician has**.
* **A musician CANNOT learn a skill** that is **NOT** inthe **list of available types** (see below).

### Methods

#### \_\_init\_\_(name: str, age: int)

In the **\_\_init\_\_** method, all the needed attributes must be set.

#### learn\_new\_skill(new\_skill: str)

Each musician can learn a new skill (one at a time):

* If the **new skill is not in the skills available for the type of musician**, raise a **ValueError** with the message **"{new skill} is not a needed skill!"**
* If the new skill is already learned, raise an **Exception** with the message **"{new skill} is already learned!"**
* If everything is okay, **return the following message**: **"{musician name} learned to {new skill}."**

## 2. Class Singer

In the **singer.py** file, the class **Singer** should be implemented. The singer is a **type of musician**. The skills a singer can **learn** are:

* **"sing high pitch notes"**
* **"sing low pitch notes"**

### Methods

The class should have the following attributes:

#### \_\_init\_\_(name: str, age: int)

In the **\_\_init\_\_** method, all the needed attributes must be set.

#### learn\_new\_skill(new\_skill: str)

Add the new skill to the singer's skills if **the skill is valid and has not been learned already**.

## 3. Class Drummer

In the **drummer.py** file, the class **Drummer** should be implemented. The drummer is a **type of musician**. The skills a drummer can **have or learn** are:

* **"play the drums with drumsticks"**
* **"play the drums with drum brushes"**
* **"read sheet music"**

### Methods

The class should have the following attributes:

#### \_\_init\_\_(name: str, age: int)

In the **\_\_init\_\_** method, all the needed attributes must be set.

#### learn\_new\_skill(new\_skill: str)

Add the new skill to the drummer's skills if **the skill is valid and has not been learned already**.

## 4. Class Guitarist

In the **guitarist.py** file, the class **Guitarist** should be implemented. The guitarist is a **type of musician**. The skills a guitarist can **have or learn** are:

* **"play metal"**
* **"play rock"**
* **"play jazz"**

### Methods

The class should have the following attributes:

#### \_\_init\_\_(name: str, age: int)

In the **\_\_init\_\_** method, all the needed attributes must be set.

#### learn\_new\_skill(new\_skill: str)

Add the new skill to the guitarist's skills if **the skill is valid and has not been learned already**.

## 5. Class Band

In the **band.py** file, the class **Band** should be implemented.

### Structure

The class should have the following attributes:

* **name: str**
* A string that represents the **name of the band**.
* If the name is **an empty string**, raise a ValueError with the message: **"Band name should contain at least one character!"**
* **members: list**
* An empty list that will contain the **members (musician objects) of the band**.

### Methods

#### \_\_init\_\_(name: str)

* In the **\_\_init\_\_** method, all the needed attributes must be set.

#### \_\_str\_\_()

The method returns the following string: **"{name of the band} with {total number of members} members."**

## 6. Class Concert

In the **concert.py** file, the class **Concert** should be implemented.

### Structure

The class should have the following attributes:

* **genre: str**
* A string that represents the **genre of the concert**.
* If the **genre is not** **"Metal", "Rock", or "Jazz"**, raise a ValueError with the message: **"Our group doesn't play {genre}!"**
* **audience: int**
  + An integer that represents the **number of people that will attend the concert**.
  + If the **audience count is less than 1**, raise a **ValueError** with the message: **"At least one person should attend the concert!"**
* **ticket\_price: float**
* A float number that represents the **price of ONE ticket**.
* If the **ticket price is less than 1.0**, raise a ValueError with the message: **"Ticket price must be at least 1.00$!"**
* **expenses: float**
* A float number that represents the **price for all expenses for the concert**.
  + If the **expenses are less than 0.00**, raise a **ValueError** with the message: **"Expenses cannot be a negative number!"**
* **place: str**
* A string that represents the **place where the concert will be performed**.
* If the place is **less than 2 chars long** or has **only white spaces**, raise a ValueError with the message: **"Place must contain at least 2 chars. It cannot be empty!"**

### Methods

#### \_\_init\_\_(genre: str, audience: int, ticket\_price: float, expenses: float, place: str)

In the **\_\_init\_\_** method, all the needed attributes must be set.

#### \_\_str\_\_()

The method returns the following string: **"{genre} concert at {place}."**

### 7. Class ConcertTrackerApp

In the **concert\_tracker\_app.py** file, the class **ConcertTrackerApp** should be implemented. It will contain all the functionality of the project.

### Structure

The class should have the following attributes:

* **bands: list**
  + An **empty** list that will contain **all the bands** (objects).
* **musicians: list**
  + An **empty** list that will contain **all the musicians** (objects).
* **concerts: list**
  + An **empty** list that will contain **all the concerts** (objects).

### Methods

#### \_\_init\_\_()

In the **\_\_init\_\_** method, all the needed attributes must be set.

#### create\_musician(musician\_type: str, name: str, age: int)

The method **creates a new musician**.

* Valid musician types: **"Guitarist", "Drummer", "Singer"**
* If the musician type is not a valid type, raise a **ValueError** with the message **"Invalid musician type!"**
* If there is a musician with the **same name**, raise an **Exception** with the message **"{musician\_name} is already a musician!"**
* If everything is valid, **create** the musician, **add** it to the musicians' list**,** and **return** a message **"{musician\_name} is now a {musician\_type}."**

#### create\_band(name: str)

The method **creates a new band**.

* If **there is already a band with the same name**, raise an **Exception** with the message **"{band\_name} band is already created!"**
* If everything is valid, **create a new band, add** it to the bands' list**,** and return a message **"{band\_name} was created."**

#### create\_concert(genre: str, audience: int, ticket\_price: float, expenses: float, place: str)

* If **there is already a concert in the same place,** raisean **Exception** with the message **"{concert\_place} is already registered for {concert\_genre} concert!"**
* If everything is valid, **create a new concert, add** it to the concerts' list**,** and return a message **"{concert\_genre} concert in {concert\_place} was added."**

#### add\_musician\_to\_band(musician\_name: str, band\_name: str)

The method **adds a musician** to the band.

* If **there isn't a musician with the given name**, raise an **Exception** with the message **"{musician\_name} isn't a musician!"**
* If **there isn't a band with the given name**, raise an **Exception** with the message **"{band\_name} isn't a band!"**
* If everything is valid, **add the musician to the band** and return the message **"{musician\_name} was added to {band\_name}."**

#### remove\_musician\_from\_band(musician\_name: str, band\_name: str)

The method **removes a musician** from the band.

* If **there isn't a band with the given name**, raise an **Exception** with the message **"{band\_name} isn't a band!"**
* If **there isn't a musician with the given name who is a member of the given band**, raise an **Exception** with the message **"{musician\_name} isn't a member of {band\_name}!"**
* If everything is valid, **remove the musician from the band** and return the message **"{musician\_name} was removed from {band\_name}."**

#### start\_concert(concert\_place: str, band\_name: str)

The method is to start a concert at the given place with the given band. The concert **place** and the band **name** will **always be valid**. However, there are some **rules** for the band to start a concert **depending on the band members** and the **concert type**:

* If there **is NOT at least one member of each type** (at least 1 singer, at least 1 drummer, and at least 1 guitarist), raise an **Exception** with the message **"{band name} can't start the concert because it doesn't have enough members!"**
* **Then, check if the band can play at the concert**:
  + For a band to play at a **"Rock" concert**, the **needed skills for** **all members** depending on the musician type are:
    - **Drummer - play the drums with drumsticks**
    - **Singer - sing high pitch notes**
    - **Guitarist – play rock**
  + For a band to play at a **"Metal" concert**, the **needed skills for** **all members** depending on the musician type are:
    - **Drummer - play the drums with drumsticks**
    - **Singer - sing low pitch notes**
    - **Guitarist – play metal**
  + For a band to play at a **"Jazz" concert**, the **needed skills for** **all members** depending on the musician type are:
    - **Drummer - play the drums with drum brushes**
    - **Singer - sing high pitch notes** and **sing low pitch notes**
    - **Guitarist – play jazz**
* If **one or more band members don't have the required skills** to play at a concert, **raise an Exception** with the message **"The {band\_name} band is not ready to play at the concert!"**
* If **all members can play at a concert**, **calculate the profit** by the following formula: **"(audience \* ticket price) - expenses"**, and return the following message: **"{band\_name} gained {profit}$ from the {concert\_genre} concert in {concert\_place}."**
  + Profit should be formatted to the **second decimal place**.

## Examples

|  |
| --- |
| **Input** |
| from project.concert\_tracker\_app import ConcertTrackerApp  musician\_types = ["Singer", "Drummer", "Guitarist"]  names = ["George", "Alex", "Lilly"]  app = ConcertTrackerApp()  for i in range(3):  print(app.create\_musician(musician\_types[i], names[i], 20))  print(app.musicians[0].learn\_new\_skill("sing high pitch notes"))  print(app.musicians[1].learn\_new\_skill("play the drums with drumsticks"))  print(app.musicians[2].learn\_new\_skill("play rock"))  print(app.create\_band("RockName"))  for i in range(3):  print(app.add\_musician\_to\_band(names[i], "RockName"))  print(app.create\_concert("Rock", 20, 5.20, 56.7, "Sofia"))  print(list(map(lambda a: a.\_\_class\_\_.\_\_name\_\_, app.bands[0].members)))  print(app.start\_concert("Sofia", "RockName")) |
| **Output** |
| George is now a Singer.  Alex is now a Drummer.  Lilly is now a Guitarist.  George learned to sing high pitch notes.  Alex learned to play the drums with drumsticks.  Lilly learned to play rock.  RockName was created.  George was added to RockName.  Alex was added to RockName.  Lilly was added to RockName.  Rock concert in Sofia was added.  ['Singer', 'Drummer', 'Guitarist']  RockName gained 47.30$ from the Rock concert in Sofia. |

# Problem 3. Unit Tests

You will **be provided with another skeleton** for this problem. **Open** the **new skeleton** as a **new project** and write tests for the **TruckDriver** class. The class will have some methods, fields, and one constructor, all of them working properly. You are **NOT ALLOWED** to change any class. Cover the whole class with unit tests to make sure that the class is working as intended. Submit **only the test** folder.